CloudMdsQL: querying heterogeneous cloud data stores with a common language

Boyan Kolev[1]· Patrick Valduriez[1] ·
Carlyna Bondiombouy[1] · Ricardo Jiménez-Peris[2] ·
Raquel Pau[3] · José Pereira[4]

Abstract The blooming of different cloud data management infrastructures, specialized for different kinds of data and tasks, has led to a wide diversification of DBMS interfaces and the loss of a common programming paradigm. In this paper, we present the design of a cloud multidatastore query language (CloudMdsQL), and its query engine. CloudMdsQL is a functional SQL-like language, capable of querying multiple heterogeneous data stores (relational and NoSQL) within a single query that may contain embedded invocations to each data store's native query interface. The query engine has a fully distributed architecture, which provides important opportunities for optimization. The major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph database) to be called as functions, and at the same time be optimized, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping. Our experimental validation, with three data stores (graph, document and relational) and representative queries, shows that CloudMdsQL satisfies the five important requirements for a cloud multidatastore query language.

✉ Boyan Kolev
boyan.kolev@inria.fr

Patrick Valduriez
patrick.valduriez@inria.fr

Carlyna Bondiombouy
carlyna.bondiombouy@inria.fr

1 Zenith team, Inria, Montpellier, France

2 Universidad Politecnica de Madrid (UPM) and LeanXcale, Madrid, Spain

3 Sparsity Technologies, Barcelona, Spain

4 INESC, Braga, Portugal

1 Introduction

A major trend in cloud computing and data management is the understanding that there is "no one size fits all" solution. Thus, there has been a blooming of different cloud data management infrastructures, referred to as NoSQL [20], specialized for different kinds of data and tasks and able to perform orders of magnitude better than traditional relational DBMS. Examples of new data management technologies include: graph databases (e.g. Sparksee [18], Neo4j), key-value data stores (e.g. HBase, Cassandra, HyperTable), array data stores (e.g. SciDB), analytical cloud databases (e.g. Greenplum and Vertica), analytical cloud frameworks (e.g. Hadoop Map-Reduce, Cloudera Impala), document databases (e.g. MongoDB, CouchBase), and data stream management systems (e.g. StreamCloud [9,10], Storm). This has resulted in a rich offering of services that can be used to build cloud data-intensive applications that can scale and exhibit high performance. However, this has also led to a wide diversification of DBMS interfaces and the loss of a common programming paradigm.

This makes it very hard for a user to integrate her data sitting in specialized data stores, e.g. relational, documents and graph databases. For example, consider a user who, given a relational data store with authors, a document store with reviews, and a graph database with author friendships, wants to find out about conflicts of interests in the reviewing of some papers. The main solution today would be to write a program (e.g. in Java) that accesses the three data stores through their APIs and integrates the data (in memory). This solution is obviously labor-intensive, complex and not easily extensible (e.g. to deal with a new data store).

The CoherentPaaS project [5] addresses this problem, by providing a rich Platform as a Service (PaaS) with different "one size" systems optimized for particular tasks, data and workloads. However, unlike in the current cloud landscape, it provides a common programming model and language to query multiple data stores. The platform is designed to allow different subsets of enterprise data to be materialized within different data models, so that each subset is handled in the most efficient way according to its most common data access patterns. On the other hand, an application can still access a data store directly, without using our query engine. This constitutes a multidatastore system with high levels of heterogeneity and local autonomy. In this paper, we focus on the problem of querying heterogeneous cloud data stores (in read-only mode) with a common language.

The problem of accessing heterogeneous data sources, i.e. managed by different data management systems such as relational DBMS or XML DBMS, has long been studied in the context of multidatabase systems [21] (also called data integration systems in the context of the web [7]). However, the state-of-the-art solutions for multidatabase systems (see Sect. 2) do not directly apply to solve our problem. First, our common language is not for querying data sources on the web, which could be in very high numbers. A query should be on a few cloud data stores (perhaps less than 10) and the user needs to have access rights to each data store. Second, the data stores

may have very different languages, ranging from very simple get/put in key-value stores, to full SQL or SPARQL languages. And no single language can capture all the others efficiently, e.g. SQL cannot express graph path traversal (of course, we can represent a graph with two relations Edges and Nodes, but this requires translating path traversals into expensive joins). Even a graph query language, which is very general, cannot capture an array data model easily. Third, NoSQL databases can be without schema, which makes it (almost) impossible to derive a global schema. Finally, and very important, what the user needs is the ability to express powerful queries to exploit the full power of the different data store languages, e.g. directly express a path traversal in a graph database. For this, we need a new query language.

We can translate these observations into five main requirements for our common language:

1. To integrate fully-functional queries against different NoSQL and SQL databases using each database's native query mechanism;
2. To allow nested queries to be arbitrarily chained together in sequences, so the result of one query (for one database) may be used as the input of another (for another database);
3. To be schema independent, so that databases without or with different schemas can be easily integrated;
4. To allow data-metadata transformations, e.g. to convert attributes or relations into data and vice versa [24];
5. To be easily optimizable so that efficient query optimization, introduced in state-of-the-art multidatabase systems, can be reused (e.g. exploiting bind joins [11] or shipping the smallest intermediate results).

In this paper, we present the design of a Cloud multidatastore query language (CloudMdsQL), and its query engine, which addresses these requirements. While the latter four have already been identified as requirements and introduced in multidatabase mediator/wrapper architectures, CloudMdsQL contributes to satisfying also the first one. The language is capable of querying multiple heterogeneous databases (e.g. relational and NoSQL) within a single query containing nested subqueries, each of which refers to a particular data store and may contain embedded invocations to the data store's native query interface.

The design of the query engine takes advantage of the fact that it operates in a cloud platform. Unlike the traditional mediator/wrapper architectural model where mediator and wrappers are centralized, we propose a fully distributed architecture that yields important optimization opportunities, e.g. minimizing data transfers between nodes. This allows us to reuse query decomposition and optimization techniques from distributed query processing [21]. Thus, the major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph database) to be called as functions, and at the same time be optimized based on a simple cost model, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping.

The rest of this paper is organized as follows. Section 2 discusses related work in more details. Section 3 introduces CloudMdsQL's basic concepts, including its data

model and language constructs. Section 4 presents the architecture of the query engine and its main components. Section 5 presents the language in more details. Section 6 reveals the query processing steps. Section 7 gives an example walkthrough. Section 8 presents an experimental validation, with three data stores: Sparksee (graph database), MongoDB (documents database) and Derby (relational database). Section 9 concludes.

2 Related work

Accessing heterogeneous data sources has been addressed by multidatabase systems [21] and data integration systems for the Web [7]. The typical solution is to provide a common data model and query language to transparently access data sources, thus hiding data source heterogeneity and distribution. The dominant state-of-the-art architectural model is the mediator/wrapper architecture. In this architecture, each data source has an associated wrapper that exports information about the source schema and data, and mapping functions that give the translation between source data and schemas and the mediator's data and schema. The mediator centralizes the information provided by the wrappers in a unified view (called mediated schema) of all available data, transforms queries expressed in a common language into queries for the data sources using the wrappers, and integrates the queries' results.

The mediator/wrapper architecture has several advantages. First, the specialized components of the architecture allow the various concerns of different kinds of users to be handled separately. Second, mediators typically specialize in a related set of data sources with "similar" data, and thus export schemas and semantics related to a particular domain. The specialization of the components leads to a flexible and extensible distributed system. In particular, it allows seamless integration of different data stored in very different data sources, ranging from full-fledged relational databases to simple files. The authors of [22] propose distributed mediator architecture with a flexible interface between mediators and data sources that efficiently handles different query languages and different data source functionality. The wrapper interface provides the mediator awareness of the capabilities of each data source according to the common data model.

The common data model and query language used by the mediator have a major impact on the effectiveness of data source integration. The two dominant solutions today, with major product offerings, are relational/SQL and XML/Xquery, each having its own advantages. The relational model provides a simple data representation (tables) for mapping the data sources, but with rigid schema support. The major advantage of a relational solution is that SQL is familiar to users and developers, with SQL APIs used by many tools, e.g. in business intelligence. Furthermore, recent extensions of SQL such as SQL/XML include support for XML data types. On the other hand, the XML model provides a tree-based representation that is appropriate for Web data, which are typically semi-structured, and flexible schema capabilities. As a particular case, XML can represent relational tables, but at the expense of more complex parsing. XQuery is now a complete query language for XML, including update capabilities, but more complex than SQL. As a generalization for Web linked data, there is also current work based on RDF/SPARQL [12]. There is still much debate on relational versus XML, but

in the cloud, relational-like data sources, e.g. NoSQL key-value stores such as Google Bigtable and Hadoop Hbase, are becoming very popular, thus making a relational-like model attractive.

The main requirements for a common query language (and data model) are support for nested queries, schema independence and data-metadata transformation [24]. Nested queries allow queries to be arbitrarily chained together in sequences, so the result of one query (for one data store) may be used as the input of another (for another data store). Schema independence allows the user to formulate queries that are robust in front of schema evolution. Data-metadata transformation is important to deal with heterogeneous schemas by transforming data into metadata and conversely, e.g. data into attribute or relation names, attribute names into relation names, relation names into data. These requirements are not supported by query languages designed for centralized databases, e.g. SQL and XQuery. Therefore, federated query languages need major extensions of their centralized counterpart.

We now discuss briefly two kinds of such extensions of major interest: relational languages and functional SQL-like languages. In [24], the authors propose an extended relational model for data and metadata integration, the Federated Relational Data Model, with a relational algebra, Federated Interoperable Relational Algebra (FIRA) and an SQL-like query language that is equivalent to FIRA, Federated Interoperable Structured Query Language (FISQL). FIRA and FISQL support the requirements discussed above, and the equivalence between FISQL and FIRA provides the basis for distributed query optimization. FISQL and FIRA appear as the best extensions of SQL-like languages for data and metadata integration. In particular, it allows nested queries. But as with SQL, it is not possible to express some complex control on how queries are nested, e.g. using programming language statements such as IF THEN ELSE, or WHILE. Note that, to express control over multiple SQL statements, SQL developers typically rely on an imperative language such as Java in the client layer or a stored procedure dialect such as PLSQL in the database layer. Another major limitation of the relational language approach is that it does not allow exploiting the full power of the local data source repositories. For instance, mapping an SQL-like query to a graph database query will not exploit the graph DBMS capabilities, e.g. generating a breadth-first search query.

Database programming languages (DBPLs) have been proposed to solve the infamous impedance mismatch between programming language and query language. In particular, functional DBPLs such as FAD [6] can represent all query building blocks as functions and function results can be used as input to subsequent functions, thus making it easy to deal with nested queries with complex control. The first SQL-like functional DBPL is Functional SQL [23]. More recently, FunSQL [2] has been proposed for the cloud, to allow shipping the code of an application to its data. Another popular functional DBPL is LINQ [19], whose goal is to reconcile object-oriented programming, with relations and XML. LINQ allows any .NET programming language to manipulate general query operators (as functions) with two domain-specific APIs that work over XML (XLinq) and relational data (DLinq) respectively. The operators over relational data provide a simple object-relational mapping that makes it easy to specify wrappers to the underlying RDBMS. More recently, in the context of the cloud, Spark SQL [1] has been proposed as an Apache Spark module to provide tight

integration between relational and procedural processing through a declarative API that integrates relational operators with procedural Spark code, taking advantage of massive parallelism. Similarly to LINQ, Spark SQL can map to relations arbitrary Java objects as well as different data sources. In addition, it includes a flexible and extensible optimizer that supports operator pushdowns to data sources, according to their capabilities.

Recently, multistore systems have been introduced to provide integrated access to a number of RDBMS and NoSQL data stores through a common query engine. The BigIntegrator system [26] integrates data from cloud-based NoSQL, such as Google's Bigtable, and relational databases. The system relies on mapping a limited set of relational operations to native queries expressed in GQL (Google Bigtable query language). With GQL, the task is achievable because it represents a subset of SQL. However, unlike CloudMdsQL, it only works for Bigtable-like systems and cannot integrate data from other families of NoSQL systems, e.g. document or graph databases. Estocada [4] is a self-tuning multistore platform that uses view-based rewriting for providing access to datasets in native format while automatically placing fragments of the datasets across heterogeneous stores. Since these approaches do not directly support native queries, they do not preserve the expressivity of an arbitrary data store's query language.

Tightly-coupled multistore systems have been introduced with the goal of integrating Hadoop MapReduce for big data analysis with traditional RDMS with NoSQL data stores. Major multistore products such as IBM BigInsights, Microsoft HDInsight, Oracle Bigdata Appliance, typically rely on database connectors (e.g. JDBC drivers) and multidatabase techniques to integrate diverse data. Odyssey [13] and MISO [16] go one step further by addressing materialization of data across NoSQL and relational data stores and physical tuning, aiming at optimal data materialization. With MISO for instance, an application queries the execution layer with an SQL-like API, then the relational store retrieves data from both its data warehouse and the Hadoop store via materialized views. JEN [25] is another multistore system that allows joining data from two data stores, HDFS and RDBMS, with parallel join algorithms, in particular, an efficient zigzag join algorithm, and techniques to minimize data movement. As the data size grows, executing the join on the HDFS side appears to be more efficient. These systems are characterized by the absence of data store autonomy and the limited capability of integrating diverse set of data models, which distinguishes them from CloudMdsQL.

To summarize, a functional language has several advantages for accessing heterogeneous data sources. First, nested queries and complex control can be easily supported. Second and more important, the full power of the local data source repositories could be exploited, by simply allowing local data source queries, e.g. a breadth-first search query, to be called as native functions. However, DBPLs are also full-fledge programming languages, aimed to develop complex data-intensive applications. This generality makes them hard to optimize [14]. But for accessing heterogeneous data stores in the cloud, we do not need a full-fledge DBPL. Therefore, CloudMdsQL is a functional SQL-like language with minimal capabilities to access heterogeneous cloud data stores in the most efficient way, e.g. by exploiting the full power of the local data stores.

3 Basic concepts

The common querying model targets integration of data from several sources based on a diverse set of data models, mediating the data through a common data model. Consequently, the common query language and its data model are designed to achieve such data integration accordingly.

3.1 Data model

CloudMdsQL sticks to the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations. To be robust against schema evolution and driven by the fact that NoSQL data stores can be schema-less, CloudMdsQL keeps its common data model schema-less, while at the same time it is designed to ensure that all the datasets retrieved from the data stores match the common data model.

3.1.1 Operators

The common data model supports basic relational operators (projection, selection, joins, aggregation, sorting, set operations). In addition, in order to satisfy the common language requirements, the data model includes another two operators as follows. First, to support data and data-metadata transformations, CloudMdsQL introduces a Python operator that can perform user-defined operations over intermediate relations and/or generate synthetic data by executing embedded code of the programming language Python. Second, the requirement for running optimal nested queries from heterogeneous data stores implies the usage of the bind join method [11], which uses the data retrieved from one data store to rewrite the query to another data store, so that from the latter are retrieved only the tuples that match the join criteria.

3.1.2 Data types

The CloudMdsQL data model supports a minimal set of data types, enough to capture data types supported by the data models of most data stores: scalar data types—integer, float, string, binary, timestamp; composite data types—array, dictionary (associative array); null values. Standard operations over the above data types are also available: arithmetic operations, concatenation and substring, as well as operations for addressing elements of composite types (e.g. array[index] and dictionary['key']). Type constructors for the composite data types follow the well-known JSON-style constructors: an array is expressed as a comma separated list of values, surrounded by brackets; a dictionary is expressed as a comma separated list of key:value pairs, surrounded by curly braces.

3.2 Language concepts

The CloudMdsQL language itself is SQL-based with the extended capabilities for embedding native queries to data stores and embedding procedural language con-

structs. The involvement of the latter is necessitated by the requirement for performing data and data-metadata transformations and conversions of arbitrary datasets to relations in order to comply with the common data model. To support such procedural programmability, CloudMdsQL queries can contain embedded constructs of the programming language Python, the choice of which is justified by its richness of data types, native support for iteration with generator functions, ease of use, richness in standard libraries and wide usage.

An important concept in CloudMdsQL is the notion of "table expression", inspired from XML table views [8,17], which is generally an expression that returns a table (i.e. a structure, compliant with the common data model). A table expression is used to represent a nested query and usually addresses a particular data store. Three kinds of table expressions are distinguished:

- Native table expressions, using a data store's native query mechanism;
- SQL table expressions, which are regular nested SELECT statements;
- Embedded blocks of Python statements that produce relations.

A table expression is usually assigned a name and a signature, thus turning it into a "named table expression", which can be used in the FROM clause of the query like a regular relation. Named table expression's signature defines the names and types of the columns of the returned relation. Thus, each CloudMdsQL query is executed in the context of an ad-hoc schema, formed by all named table expressions within the query. This approach fills the gap produced by the lack of a global schema and allows the query compiler to perform semantic analysis of the query. A named table expression is usually defined as a query against a particular data store and contains references to the data store's data structures. However, the expression can also instantiate other named table expressions, defined against other data stores, thus chaining data as per the requirement for nesting queries.

For example, the following simple CloudMdsQL query contains two subqueries, defined by the named table expressions T1 and T2, and addressed respectively against the data stores DB1 (an SQL database) and DB2 (a MongoDB database):

```
T1(x int, y int)@DB1 = ( SELECT x, y FROM A )
T2(x int, z string)@DB2 = {*
  db.B.find( {$lt: {x, 10}}, {x:1, z:1, _id:0} )
*}
SELECT T1.x, T2.z
FROM T1, T2
WHERE T1.x = T2.x AND T1.y <= 3
```

The purpose of this query is to perform relational algebra operations (expressed in the main SELECT statement) on two datasets retrieved from a relational and a document database. The two subqueries are sent independently for execution against their data stores in order the retrieved relations to be joined by the common query engine. The SQL table expression `T1` is defined by an SQL subquery, while `T2` is a native expression using a MongoDB query that retrieves from the document collection B the attributes `x` and `z` of those documents for which `x < 10`. The subquery of expression `T1` is subject to rewriting by pushing into it the filter condition `y <= 3`, specified in the main SELECT statement, thus reducing the amount of the retrieved

data by increasing the subquery selectivity. The so retrieved datasets are then converted to relations following their corresponding signatures, so that the main CloudMdsQL SELECT statement can be processed with semantic correctness.

4 Query engine architecture

Although the focus of this paper is on the design of the CloudMdsQL language, we still need to show how queries can be optimized and processed in a cloud environment. Thus, in this section, we introduce the architecture of the query engine, with its main components, and briefly introduce query processing, which will be more detailed in Sects. 6 and 7. We ignore fault tolerance, which is out of the scope of this paper.

4.1 Overview

The design of the query engine takes advantage of the fact that it operates in a cloud platform, with full control over where the system components can be installed. This is quite different from web data integration systems for instance, where both mediator and data source wrappers can only be installed at one or more servers that communicate with the data sources through the network. In our context, the query engine is part of a more general platform (CoherentPaaS) that allows deployment over one or more data centers. For simplicity in this paper, we consider the case of a single data center, i.e. a computer cluster.

The architecture of the query engine is fully distributed (see Fig. 1), so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. Thus, the query engine does not follow the traditional medi-

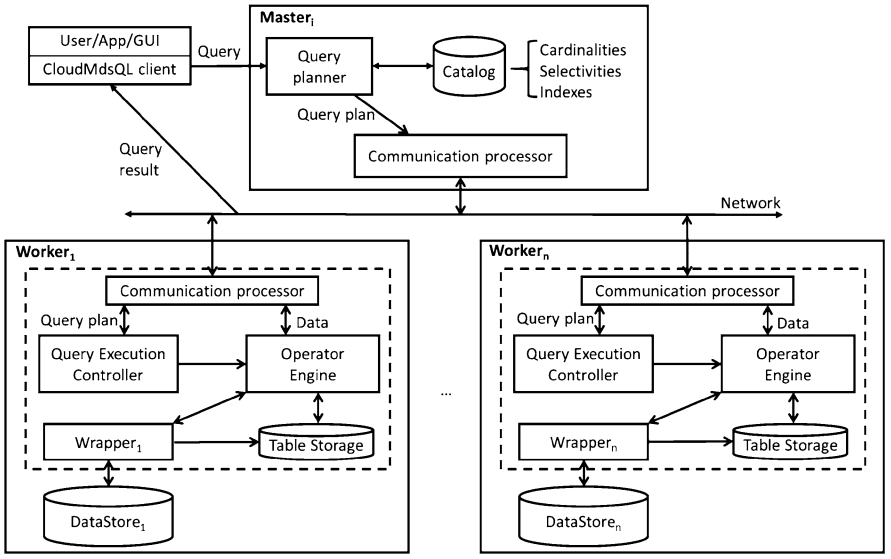


Fig. 1 Architecture of the query engine

ator/wrapper architectural model where mediator and wrappers are centralized. This distributed architecture yields important optimization opportunities, e.g. minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node.

Each query engine node consists of two parts—master and worker—and is collocated at each data store node in a computer cluster. Each master or worker has a communication processor that supports send and receive operators to exchange data and commands between nodes. To ease readability in Fig. 1, we separate master and worker, which makes it clear that for a given query, there will be one master in charge of query planning and one or more workers in charge of query execution. To illustrate query processing with a simple example, let us consider a query Q on two data stores in a cluster with two nodes (e.g. the query introduced in Sect. 3.2). Then a possible scenario for processing Q, where the node id is written in subscript, is the following:

- At client, send Q to Master_1.
- At Master_1, produce a query plan P (see Fig. 2) for Q and send it to Worker_2, which will control the execution of P.
- At Worker_2, send part of P, say P_1, to Worker_1, and start executing the other part of P, say P_2, by querying DataStore_2.
- At Worker_1, execute P_1 by querying DataStore_1, and send result to Worker_2.
- At Worker_2, complete the execution of P_2 (by integrating local data with data received from Worker_1), and send the final result to the client.

This simple example shows that query execution can be fully distributed among the two nodes and the result sent from where it is produced directly to the client, without the need for an intermediate node.

4.2 Master

Since there are multiple masters (one at each cluster node), the client chooses one of them to send a query to. Although load balancing is not crucial as masters do not carry

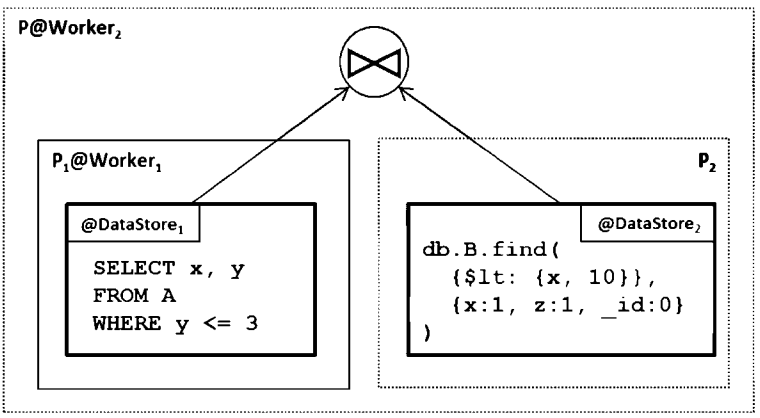


Fig. 2 A simple query plan

heavy loads, we can still do it using a random pick or a simple round robin process at the client side to distribute queries across masters.

A master takes as input a query and produces a query plan, which it sends to one chosen query engine node for execution. The query planner performs query analysis and optimization, and produces a query plan serialized in a JSON-based intermediate format that can be easily transferred across query engine nodes. Each operation in the plan carries the identifier of the query engine node that is in charge of performing it. Thus, the topmost operation determines the first worker, to which the master should send the query plan. As for declarative query languages (e.g. SQL), a query plan can be abstracted as a tree of CloudMdsQL operators and communication (send/receive) operators to exchange data and commands between query engine nodes. This allows us to reuse query decomposition and optimization techniques from distributed query processing [21], which we adapt to our fully distributed architecture. In particular, we strive to:

- Minimize local execution time in the data stores, by pushing down select operations in the data store subqueries and exploiting bind join by query rewriting;
- Minimize communication cost and network traffic by reducing data transfers between workers.

To compare alternative rewritings of a query, the query planner uses a simple catalog, which is replicated at all nodes in primary copy mode. The catalog provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Such information can be given with the help of the data store administrators. The query language provides a possibility for the user to define cost and selectivity functions whenever they cannot be derived from the catalog, mostly in the case of using native subqueries. The search space explored for optimization is the set of all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, join ordering, and intermediate data shipping. Unlike in traditional query optimization where many different permutations are possible, this search space is not very large, so we can use a simple exhaustive search strategy.

4.3 Worker

Workers collaborate to execute a query plan, produced by a master, against the underlying data stores involved in the query. As illustrated in Sect. 4.2, there is a particular worker, selected by the query planner, which becomes in charge of controlling the execution of the query plan. This worker can subcontract parts of the query plan to other workers and integrate the intermediate results to produce the final result.

Each worker node acts as a lightweight runtime database processor atop a data store and is composed of three generic modules (i.e. same code library)—query execution controller, operator engine, and table storage—and one wrapper module that is specific to a data store. These modules provide the following capabilities:

- Query execution controller: initiates and controls the execution of a query plan (received from a master or worker) by interacting with the operator engine for

local execution or with one or more workers (through communication processors) in case part of the query plan needs to be subcontracted. In the latter case, the query execution controller will synchronize the execution of the operator(s) that require the intermediate results produced by the distant workers, once they are received back.

- Operator engine: executes the query plan operators on data retrieved from the wrapper, from another worker, or from the table storage. These operators include CloudMdsQL operators to execute table expressions in the query and communication (send/receive) operators to exchange data with other workers. Some operators are simply passed on to the wrapper for producing intermediate results from the data store. The operator engine may write intermediate relations to the table storage.
- Table storage: provides efficient, uniform storage (main memory and disk) for intermediate and result data in the form of tables. Storage of intermediate data is necessary in particular cases, e.g. when an intermediate relation needs to be consumed by more than one operator or when it participates in a blocking operation such as aggregation, sorting or nested-loop join. In other cases, intermediate relations are directly pulled by the consuming operator.
- Wrapper: interacts with its data store through its native API to retrieve data, transforms the result in the form of table, and writes the result in table storage or delivers it to the operator engine. To query its data store, each wrapper is invoked by the operator engine through generic interface methods, which it maps to data store specific API calls. Wrappers are discussed in more detail in Sect. 6.4.

5 Query language

The major innovation of CloudMdsQL refers to the involvement of native subqueries and the way both SQL and native subqueries interoperate with each other to provide the desired coherence across all data stores. In this section we provide details about how multiple diverse data stores can be queried through CloudMdsQL by means of nested table expressions.

Named table expressions are definitions of temporary (at query level) tables representing nested subqueries against data stores and their signatures define the names and types of the attributes of the returned relations. Within a single CloudMdsQL query, all named table expressions form an ad-hoc schema, in the context of which the main SELECT statement of the query is processed and its semantic correctness is verified. Embedded Python constructs that can be used to define Python named table expressions necessitate the involvement of special conventions, the usage of which provides the required query expressivity and ability for nesting subqueries.

5.1 Named table expressions

Named table expressions are defined in the header of a CloudMdsQL query, preceding the main SELECT statement, and are instantiated in the FROM clause and/or from the

definitions of other named expressions. The basic syntax of a named table expression is the following:

```
<expr-name>(<colname> <type>, …)[@<datastore>] = <expr-def>
```

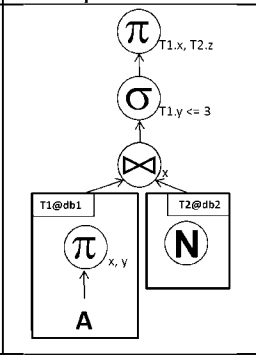
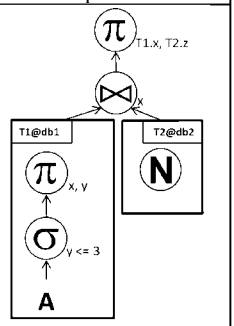
The declaration consists of the name of the expression, followed by its signature, which specifies the names and types of the attributes of the result relation, reference to the underlying data store, which the subquery is addressed to, and expression definition. An SQL expression definition should be surrounded by parentheses, which implies that the compiler processes it and transforms it to a subquery plan, part of the global execution plan, and therefore is subject to analysis, optimization and/or rewriting. A native/Python expression definition must be surrounded by native expression brackets, which is the following pair of opening / closing bracket symbols: `{*...*}`. Named table expressions are classified according to the way they interface the underlying data stores and/or intermediate relations, as follows.

Native named table expressions represent subqueries to data stores using their native query mechanism. They are executed in the context of a particular connection to a data store. The expression definition is a native query or code that should contain invocations to the native interface of the data store and produce a relation with the declared signature. The code is surrounded by native expression brackets, which gives information to the query engine not to process the contained expression but pass it as a black box to the corresponding wrapper. However, a native expression can still use as input intermediate data retrieved by other named table expressions, thus providing full capability for nesting queries. The query engine allows this by exposing the query execution context to the wrappers, like it does for Python expressions (see below).

SQL named table expressions are expressed as regular SELECT statements. They are quite different from native expressions, since they are compiled and analyzed by the query planner, as opposed to native expressions which are considered as black boxes and are not subject to analysis. An SQL expression against a data store contains in its FROM clause references to the data store tables. However, to provide support for nested querying, each SQL expression can also instantiate other named table expressions from the context of the current CloudMdsQL query (nested SQL queries are more detailed in Sect. 5.2.1). Furthermore, each data store subquery, expressed as an SQL expression, is subject to rewriting, whenever selection pushdowns or bind joins take place.

To illustrate the usage of native and SQL table expressions and give a basic notion of how they are handled by the query planner, let us come back to the example, introduced in Sect. 3.2. The CloudMdsQL query below contains an SQL named table expression `T1` and a native named table expression `T2`. The query plan after decomposition shows that the SQL expression `T1` is decomposed to a sub-plan assigned to the data store `db1`, while the sub-plan for `db2` contains only a single node, corresponding to the definition of the native expression `T2`. Thus, the sub-plan for `db1` may be modified by the planner, e.g. by pushing operations into it, as it is shown with the plan after selection pushdown. All the query processing steps are detailed in Sect. 6.

CloudMdsQL query	Query plan after decomposition	Query plan after selection pushdown

`T1(x int, y int)@db1 = (`
`  SELECT x, y FROM A`
`)`
`T2(x int, z string)@db2 = {*`
`  db.B.find( {$lt: {x, 10}},`
`  {x:1, z:1, _id:0} )`
`*}`
`SELECT T1.x, T2.z`
`FROM T1, T2`
`WHERE T1.x = T2.x`
`  AND T1.y <= 3`
		

Python named table expressions do not reference a data store and their expression definitions are surrounded by native expression brackets. When processing a Python table expression, the query compiler generates a task for Python operator, associated with the expression code, which is executed by the operator engine using an embedded Python interpreter. Such expressions can be used to perform data and metadata transformations over intermediate relations and/or generate synthetic data. CloudMdsQL provides powerful tools for interoperability between embedded Python code and the context of a query execution. First, the usage of the keyword `yield` within the Python code of an expression appends tuples to the expression's result set according to its signature. Second, a special Python object named `CloudMdsQL` that represents the context, in which the Python expression is executed, can be invoked from the Python code, thus providing access to data retrieved from data stores by other table expressions. This approach allows for a Python expression to use as input the result of other subqueries and thus provides ability for nesting subqueries as explained in more detail in the following subsection.

From query engine's point of view, the difference between Python and native expressions is that Python expressions are processed by a Python operator, which is part of the operator engine, while native expressions are delivered to the corresponding wrappers to process them. From CloudMdsQL programmer's point of view, the common point between Python and native expressions is that if a data store provides a Python API, the native expressions against it can also be written in Python and can also make use of the interoperability tools mentioned above, so that while writing a native expression, the programmer can benefit from the same high expressivity and data integration ability that is available in Python expressions. To provide this programmability, the wrapper implementation can reuse the CloudMdsQL framework for embedding Python expressions. However, if the data store does not provide Python query interface, it is the wrapper implementer's responsibility to provide mechanism to yield tuples and expose the `CloudMdsQL` object using native language's concepts.

Enhanced features of CloudMdsQL include parameterizing and storing of named expressions. In a parameterized named expression the names and types of the parameters are declared in the signature following the `WITHPARAMS` keyword. Each parameter is then referenced inside the expression by a named placeholder (e.g. the

parameter name prefixed by a dollar sign). Parameterized expressions need to be instantiated from other expressions or in a FROM clause by passing actual parameter values. CloudMdsQL also provides a `CREATE NAMED EXPRESSION` command that allows an expression to be given a global name and stored in a global catalog in order to be referenced in several queries, similarly to SQL views and stored procedures/functions.

5.2 Nested queries

As stated in the language requirements, CloudMdsQL must provide a mechanism for nesting queries—i.e. a named table expression must be able to instantiate other named table expressions, available in the execution context of the same query, and use their result sets as input. This is achievable in all types of expressions: in native/Python expressions by invoking the `CloudMdsQL` object, and in SQL expressions by simply referencing named table instantiations directly in the FROM clause, often in combination with references to the data store's tables.

5.2.1 Within SQL expressions

An SQL expression against a data store contains references to data store tables, but may also refer to named table expressions in its FROM clause. If the SQL expression contains such mixed references, its corresponding subquery plan is split by the query compiler into two sub-plans. The first one contains only references to data store tables and is identified as a subquery plan that will be passed to the wrapper. The other one references only the root node of the first sub-tree and instantiations of other named table expressions from the context of the CloudMdsQL query and will be executed by the query engine as part of the common execution plan. This is illustrated with the following example:

Original query	Rewritten equivalent query

`T1(x int, y int)@DB2 = {*`
`  db.B.find( {$lt: {x, 10}},`
`  {x:1, y:1, _id:0} )`
`*}`
`T2(x int, y int, z string)@DB1 = (`
`  SELECT A.x, T1.y, B.z`
`  FROM A JOIN B ON A.id = B.id`
`  **JOIN T1 ON A.x = T1.x**`
`)`
`SELECT x, y, z FROM T2`
	`T1(x int, y int)@DB2 = {*`
	`  db.B.find( {$lt: {x, 10}},`
	`  {x:1, y:1, _id:0} )`
	`*}`
	`T2(x int, z string)@DB1 = (`
	`  SELECT A.x, B.z`
	`  FROM A JOIN B ON A.id = B.id`
	`)`
	`SELECT T2.x, T1.y, T2.z`
	`FROM T2 **JOIN T1 ON T2.x = T1.x**`

Here the query planner, upon parsing the original query and building the subquery plan for T2, detects the usage of the named table T1, plans the join with T1 as the outermost operation within the sub-plan, and pulls it into the common plan, thus transforming the whole query plan to correspond to the rewritten equivalent query above. In some more complex cases, the planner may not be able to place as outermost all the operations that involve named table expressions; in such cases, the planner will

split the sub-plan in order to be able to pull such operations from the sub-plan, which may result in building more than one sub-plans that originate from a single subquery.

Another nested query scenario is when a named table is referred in a sub-select statement within the subquery, thus making the result set of the named table an input to the subquery, as in the following example:

```
T1(x int, y int)@DB2 ={* db.B.find({$lt: {x, 10}}, {x:1, y:1, _id:0}) *}
T2(x int, z string)@DB1 = (
  SELECT A.x, B.z FROM A JOIN B ON A.id = B.id
  WHERE A.x IN (SELECT x FROM T1 WHERE y > 0)
)
SELECT x, z FROM T2
```

To process the subquery `T2`, the query engine must first retrieve the table `T1`, evaluate the sub-select `SELECT x FROM T1 WHERE y > 0`, and then transform it to a list of the distinct values of `T1.x` to replace the sub-select with that list of values. This is similar to the processing of bind joins, which is explained in detail in Sect. 6.2.

5.2.2 Within native expressions

This subsection focuses on the capability of nesting subqueries within native/Python expressions. CloudMdsQL introduces two approaches that allow the programmer to write expression definitions that iterate through data retrieved by other subqueries—**table iteration** and **join iteration**.

With **table iteration**, the Python code of a table expression can iterate through the result set of another table expression by requesting a forward iterator through the `CloudMdsQL` object, instantiating the iterated table by its name. Because of the forward iteration pattern and due to the pipelining fashion of the query execution, the Python expression will start consuming tuples once a few tuples of the iterated table are available, without having to wait for the entire table to be retrieved. To build a relevant and adequate query execution plan, the query compiler needs to identify all dependencies between table expression, i.e. for each named expression, the engine needs to know which other named table expressions it iterates through. For native/Python expressions, since a black-box approach is used, the query engine does not perform any processing of the code; therefore the referenced inside the expression tables must be explicitly specified in the expression's signature. CloudMdsQL provides an additional `REFERENCING` clause, by which the programmer specifies that the expression definition performs iterations through a named table instantiation.

For example, let us consider the following query, assuming that DB1 is a relational database with a table `person`, containing names and addresses of persons, and DB2 is a graph database with Python API providing the function `GetShortestDistance`, which finds the shortest distance between two cities. Now we want to query both databases to retrieve persons who work in department Herault, the cities where they live and work and what is the distance between their home and work cities.

```
person_herault(name string, h_city string, w_city string)@DB1 = (
  SELECT name, home_city, work_city
  FROM person p
  WHERE work_dept = 'Herault'
)
person_distance(name string, h_city string, w_city string, distance int
  REFERENCING person_herault)@DB2 =
{*
  for (n, hc, wc) in CloudMdsQL.person_herault:
    yield ( n, hc, wc, GetShortestDistance(hc, wc) )
*}
SELECT name, h_city, w_city, distance FROM person_distance;
```

The execution flow of the above query is quite straightforward. It contains specialized subqueries which are chained in a strict way – first the table `person_herault` is retrieved for persons who work in Herault; then its dataset is used as input to the other subquery, the result of which is the table `person_distance` that contains one more column retrieved from the graph database by calling its function `GetShortestDistance`; and finally a projection in the main SELECT statement defines the format of the result table. This approach provides good functionality because it allows arbitrary chaining of data across subqueries. But it tends to involve less flexible queries, because it does not allow selection pushdown, and hence requires specialized subqueries like `person_herault`, where the filter condition must be specified in the subquery.

The **join iteration** approach is applicable for any native/Python table expression that is one of the sides of an equijoin. The query execution requires that the other side of the join (we will call it "the outer relation") is evaluated first, so that the native/Python expression can generate its tuples by iterating through the values of the join attribute(s) of the outer relation. Thus, only tuples that match the join criteria are generated. This approach also allows for a native/Python subquery to use as input the result set of another subquery, but in a different way – in combination with a join operation. For example, the results from the above query can be retrieved using join iteration by the following query:

```
person(name string, h_city string, w_city string, w_dept string)@DB1 = (
  SELECT name, home_city, work_city, work_dept
  FROM person p
)
distance(city_1 string, city_2 string, distance int
  JOINED ON city_1, city_2)@DB2 =
{*
  for (c1, c2) in CloudMdsQL.Outer:
    yield ( c1, c2, GetShortestDistance(c1, c2) )
*}
SELECT p.name, p.h_city, p.w_city, d.distance
FROM person p JOIN distance d
  ON p.h_city = d.city_1 AND p.w_city = d.city_2
WHERE p.w_dept = 'Herault';
```

The first thing to notice here is that the subqueries are more generic – the table expression `person` represents a projection over relational data without filters; the table expression `distance` defines a relation where each tuple consists of a pair of cities and the distance between them. And the whole query is more manipula-

ble, because the filter condition `w_dept = 'Herault'` is specified in the main SELECT statement, but it can be pushed down into the subquery. Thus, if the two named table expressions are stored in the global catalog, they can be reused in a wider range of queries.

The JOINED ON clause in the signature of the Python expression declares that whenever the table `distance` participates in an equijoin with another relation on the attributes specified in the clause, the expression will generate its tuples by iterating through the values of the join attributes of the outer relation. The REFERENCING clause is used to specify the alias, which will be used by the Python code to access the outer relation. The query is processed as follows. First, the subquery against DB1 is rewritten by adding the condition `work_dept = 'Herault'` and removing `work_dept` from the projection (it is not needed for the execution of the common query plan). Then, the subquery is executed and the query engine starts retrieving from DB1 tuples that form the result set of the outer relation. Then, the wrapper of DB2 starts the execution of the Python code that queries the graph database. It consumes a projection on the attributes `h_city` and `w_city` of the outer relation, iterating through it via the special iterator object `CloudMdsQL.Outer`, and generates the tuples of its own result set.

To handle join iteration, the operator engine pipelines the join attribute values of the outer relation to the iterator object, which allows for the native/Python expression to start immediately iterating through them as soon as a few tuples are available, without having to wait for the entire outer relation to be retrieved. Once a tuple is generated by the native/Python expression, the operator engine immediately joins it with its corresponding tuple from the outer relation, thus performing the join on-the-fly with minimal cost. During the join execution, a hash map is maintained, where each already iterated join attribute value is mapped to zero or more tuples generated by the native/Python expression. Thus, the iteration is performed over a set of distinct values of the join attribute(s) of the outer relation, which saves from duplicate invocations of native API functions that can be expensive (e.g. `GetShortestDistance`).

5.3 CloudMdsQL SELECT statement

SELECT queries in CloudMdsQL retrieve data from several data stores using embedded subqueries (for each data store) and integrate the data to build the result dataset. The CloudMdsQL SELECT statement looks like a typical SQL SELECT statement but supplements it with a header containing definitions of named table expressions:

```
[<named-table-expr> ...]
SELECT <column_list>
<from_clause>
[<where_clause>]
[<group_clause>]
[<having_clause>]
[<order_clause>]
[<limit_clause>]
```

Some of the clauses have CloudMdsQL specifics. [< named − table − expr > ...] is an optional list of named table expressions as per the corresponding syntaxes described above. Names of table expressions must be unique within both the local (in the same query) and global (stored named expressions) context. The generic syntax of a named table expression definition is presented below.

```
<expr-name>(<colname> <type>, …
            [WITHPARAMS <paramname> <type>, …]
            [REFERENCING <tablename>, …]
            [JOINED ON <colname>, …]
            [CARDINALITY = <cardinality_function>]
            [SELECTIVITY(<colname>, …) = <selectivity_function>]
)[@<datastore>] = <expr-def>
```

Its signature may contain certain optional clauses, as follows. The `WITHPARAMS` clause specifies the names and types of the parameters, if any. The `REFERENCING` clause specifies the names of other named table expressions that are used within a native named table expression. The `JOINED ON` clause specifies the names of the columns of the table expression on which a join iteration method will be performed. The `CARDINALITY` clause specifies a user-defined cost function that can be used by the optimizer to estimate the expected cardinality of the named expression's result set. The function is expressed as an arithmetic expression that may refer to the cardinalities of the referenced named tables, e.g. `card(T1)`, and/or any of the named table expression's parameters. Similarly, a `SELECTIVITY` function may also be defined, which can give an estimate of the expected selectivity of an equality condition on the specified columns.

< `from_clause` > is a regular SQL FROM clause containing references to named table expressions—global or ad-hoc, parameterized or not. If a table refers to a parameterized expression, parameter values should be specified in parentheses. The FROM clause may contain JOIN expressions, specifying explicit join ordering and conditions. The JOIN keyword may be followed or preceded by execution directive in parentheses, which will override optimizer's decisions and will explicitly make the query engine perform a concrete method (e.g. bind join, hash, merge or nested-loop).

In the < `where_clause` > there can be specified a filter predicate expression. The query compiler will transform it to normal conjunctive form, thus determining the exact selection operations to be performed as part of the execution plan. The optimizer will then find the most appropriate place of each selection operation and push it down as much as possible in the execution plan tree. This optimization can finally result in rewriting subqueries to data stores by adding filter conditions, if the optimizer finds an opportunity to increase the selectivity of the subquery. However, only subqueries defined with SQL named table expressions can benefit from such an optimization.

6 Query processing

In this section, we briefly describe in more detail the different steps of CloudMdsQL query processing, according to the query engine architecture (see Sect. 4), i.e. query decomposition, optimization and execution. We also discuss the details of interfacing

data stores through wrappers. We end with a use case example showing the different query processing steps.

6.1 Query decomposition

During query decomposition, the query planner compiles the query and builds a preliminary query execution plan (QEP). A query plan in its simplest form is a tree structure, representing relational operations, where the leaf nodes are references to tables, results from the execution of the subqueries against data stores. At this step, the planner also prepares a set of native queries which will be passed to the corresponding wrappers and hence to the underlying data stores (this process will be explained later). Each node of the query plan represents a relational operation and an intermediate relation, resulting from the operation. For more complex queries, since the language allows a single named table expression to be used as operand to several operations (e.g. referenced in other named table expressions and also in the main SELECT statement), it is possible for an intermediate relation to be the input of more than one operator, therefore the query plan appears to be a directed acyclic graph rather than a tree structure. If cyclic references exist, they will be discovered by the query engine at decomposition time and the query will be rejected.

While building the execution strategy, the planner identifies a forest of sub-trees within the query plan, each of which is associated to a certain data store. Each of these sub-plans is meant to be delivered to the corresponding wrapper, which has to translate it to a native query and execute it against its data store (for SQL subqueries, this process is more detailed in Sect. 6.4.1). The rest of the query execution plan is the part that will be handled by the query engine. So now we outline two main subsets of the global execution plan: (1) a forest of sub-trees that will be executed locally by each data store and (2) a common query plan that will be executed by the query engine with leaf nodes consuming the relations returned by each wrapper as result of sub-plan execution. At query decomposition step, the boundary between the two subsets is preliminary and may be modified during the query optimization step by pushing operations from the common plan to sub-plans to improve the overall execution efficiency or by pulling operations from sub-plans to the common plan in case a data store is not capable of handling them.

The next step of the decomposition is the semantic analysis of the query. Within only the common query plan, all table and column names are verified against the query's ad-hoc schema. On the other hand, since the query engine is agnostic to the underlying data stores' schemas, it does not perform semantic analysis of sub-plans, presuming that this will be done by the data stores upon handling each subquery's native equivalent. In fact, all the sub-plans are kept as abstract syntax trees and are never transformed into execution plans. Thus, the query engine is exempt from gathering full metadata from data stores, except those metadata needed by the optimizer, e.g. the availability of indexes and some statistics.

6.2 Query optimization

At query optimization step, the query planner generates different alternatives to the preliminary query plan and compares their costs using the cost model and the cost and

metadata information, provided by the catalog or by the user. The cost information includes cardinalities and attribute selectivities of either a whole subquery or a particular data store table. To provide as much cost information as possible, each wrapper implementation should consider the cost-estimating capability of its data store and expose cost functions following one or more of the methods below:

- For a relational data store, if the data store can efficiently estimate the cost of a subquery and the size of its result set (like EXPLAIN on prepared statements), the query planner may benefit from this to directly ask a data store through its wrapper to estimate the cost of a subquery.
- If the data store is not capable of estimating the cost of a subquery, but keeps database statistics (such as cardinalities, number of distinct values per column, etc.), the wrapper implementation should make use of all available database statistics to provide implementations of the desired cost functions.
- If none of the above methods are applicable, but the data store supports aggregate queries like COUNT(*), MIN and MAX, the wrapper should contribute to the catalog information by periodically running in background probing queries [27], thus synthesizing and keeping statistics such as the number of tuples in a table, the number of distinct values of an attribute, and the min/max values of an attribute.
- However, because of the lack of cost models in some NoSQL data stores and the limited (or lack of) capability to build database statistics, the CloudMdsQL query engine gives the wrapper developer the possibility to define custom cost functions that give default cost information in case it cannot be retrieved using the above methods. As a subject to future work, we will implement other state-of-the-art techniques for heterogeneous cost modeling [21], such as the dynamic approach, which takes into account frequently changing execution environment factors to dynamically adjust cost information.
- Finally, the user may also provide user-defined cost functions, which is particularly useful in the case of native queries. For example, the native named table expression below defines a simple cardinality function, which gives information that the estimated cardinality of the returned table will be equal to the cardinality of the `Outer` relation, over which the native expression performs iteration.

```
distance(city_1 string, city_2 string, distance int
  JOINED ON city_1, city_2
  CARDINALITY = card(Outer) )@DB2 =
{*
  for (c1, c2) in CloudMdsQL.Outer:
    yield ( c1, c2, GetShortestDistance(c1, c2) )
*}
```

With this cost information, the query optimizer executes its search strategy to transform the preliminary execution plan into an optimized one. Notice that, when building its search space, the optimizer considers all sub-plans that are assigned to data stores just as atomic leaf nodes, meaning that the operations within the sub-plans are not subject to reordering. The search space explored for optimization is the set of all possible rewritings of the initial query, by pushing down select operations, expressing bind joins, join ordering, and intermediate data shipping. The result from the optimization

step is an optimized query execution plan, where, besides the possibly modified order of common plan operations, additional information may be assigned to each operation as follows. Each binary operation (join or union) carries the identifier of the query engine node that is in charge of performing it, thus determining which intermediate relation will be shipped. Each equijoin operation carries also the join method to be performed—hash, nested-loop, merge, or bind join.

Bind join [11] is an efficient method for performing semi-joins across heterogeneous data stores that uses subquery rewriting to push the join conditions. The approach to perform a bind join is the following: the left-hand side relation is retrieved, during which the tuples are stored in an intermediate storage and the distinct values of the join attribute(s) are kept in a list of values, which will be passed as a filter to the right-hand side subquery. For example, let us consider the following CloudMdsQL query:

```
A(id int, x int)@DB1 = (SELECT a.id, a.x FROM a)
B(id int, y int)@DB2 = (SELECT b.id, b.y FROM b)
SELECT a.x, b.y FROM b JOIN a ON b.id = a.id
```

Let us assume that the query planner has decided to use the bind join method and that the join condition will be bound to the right-hand side of the join operation. First, the relation `B` is retrieved from the corresponding data store using its query mechanism. Then, the distinct values of `B.id` are used as a filter condition in the query that retrieves the relation `A` from its data store. Assuming that the distinct values of `B.id` are $b_1 \ldots b_n$, the query to retrieve the right-hand side relation of the bind join uses the following SQL approach (or its equivalent according to the data store's query language):

```
SELECT a.id, a.x FROM a WHERE a.id IN (b1, …, bn)
```

Thus, only the rows from `A` that match the join criteria are retrieved. In order to perform this operation, the final subquery to retrieve relation `A` must be composed by the query engine during runtime. Therefore, for each right-hand side of a bind join, the query compiler prepares an "almost ready" native query sentence, with placeholders for including the bind join condition, which will be added later by the query engine during runtime.

In order to estimate the expected performance gain of a bind join, the query optimizer takes into account the availability and type of indexes on the join attributes of the right-hand side relation in the data store. Whenever such information is available from the data store, the wrapper should be able to provide it. Failing to do so will prevent the planner from planning bind join, as bind joins are beneficial only in case the join attributes are indexed.

Subquery rewriting can be planned by the optimizer in several occasions: (a) selection pushdowns, which result in pushing filter conditions from the common plan to sub-trees; (b) usage of bind joins which implies adding filter conditions to the subquery in order to allow the retrieval of only those tuples that match the join criteria; (c) taking advantage of sort-merge joins which requires adding sorting operations to subqueries in order to guarantee that the retrieved relations are sorted by their join attributes. The first rewriting approach is considered always efficient, i.e. whenever

the data store is capable of handling it, the optimizer will plan selection pushdown. However, bind joins or merge joins will be planned either if explicitly specified by CloudMdsQL directives or as a result of optimization decision, of course taking into account data store's capabilities as well.

6.3 Query execution

The QEP is passed to the first worker node for execution. The query execution controller is responsible for interpreting it and controlling its execution by passing native queries to the corresponding wrappers and instructing them to deliver the retrieved datasets to the operator engine and providing the operator engine the sequence of operators it must apply to the retrieved datasets.

The execution plan is received by the query execution controller in the form of a JSON document that contains sufficient information to configure and run efficiently each of the CloudMdsQL operations. The first step of the query execution controller is to identify the sub-plans within the plan that are associated to the collocated with the worker data store. Each sub-plan is sent to its corresponding data store wrapper to be translated into a native query. Then, the query execution controller identifies the parts of the common query plan associated with other worker nodes and sends them to their corresponding query execution controllers. For the rest of the query plan, the query execution controller looks for all named tables and temporary results involved in the execution plan, identifies the dependencies and configures their behavior. Finally, the query operator must be aware of parameterized operations that can return distinct results depending on the different input parameters.

Whenever possible, relations are just pipelined as a stream of volatile tuples from one operator to another, while in other cases the results are cached inside the table storage for later use. The table storage is used to store an intermediate relation, anytime the relation cannot be directly pipelined to its consuming operator, which happens in particular cases:

- If a named table expression is used more than once within the query and thus appears an operand to more than one operator;
- If the intermediate relation is an operand to a blocking operation, such as sorting or grouping;
- If the intermediate relation is the inner side of a nested-loop or hash join.

The table storage strives to use resources efficiently—it keeps an intermediate relation in-memory unless its size becomes so big that it must be spilled to disk. The query planner takes care not to plan for storing large tables, e.g. whenever an intermediate relation with big expected cardinality is involved in a hash or nested-loop join, the query planner will assign it to the outer side of the join, thus trying to keep large tables in the pipeline stream rather than storing them, in order to avoid table storage overflows.

The operator engine is then responsible for executing the operators in the order specified by the query execution controller. When a native call is required, the operator invokes the wrapper and opens a stream of external data that is ingested into the pipeline and, optionally, cached into the table storage. When the operator requires an existing

named table, it is retrieved from the table storage and pipelined into the query execution flow. Data is never directly provided from operators to the wrapper: when necessary, the operator informs which named tables are required to solve the operation inside the data store. There is also a specific operator for CloudMdsQL that executes a Python program and pipelines the result in the form of tuples.

This iterator approach obtains tuples as soon as they are generated unless there exists a blocking operation. The resulting tuples are stored as a temporary named table into the table storage. This final table can be retrieved by the application with a forward sequential tuple iterator that supports rewinding and repositioning into marked rows. When the named table is no longer required it is automatically removed from the table storage.

6.4 Interfacing data stores

Wrappers are implemented as plugins to the query engine. In order for a data store to be accessed through the query engine, the wrapper developer must implement the corresponding wrapper following a common interface that is used by the query processor to interact with all wrappers. Whenever a CloudMdsQL query is processed, the query engine prepares a set of native subqueries (or subquery plans) that need to be executed against the data stores. The engine then passes each subquery to the corresponding wrapper, which is responsible for the following:

- The execution of native subqueries against the data store, for which there are two possibilities: (1) Server-side execution: The wrapper passes the query to the data store for remote execution (e.g. SQL); (2) Client-side execution: The wrapper executes the query locally, accessing the data store through a client library and API (e.g. Sparksee and its Python API);
- To guarantee that the retrieved data matches the number and types of columns, specified in the signature of the expected dataset in the CloudMdsQL query;
- To deliver the tuples of the retrieved datasets to the operator engine;
- To be able to instantiate other named table expressions, hence to access intermediate relations from the operator engine (table storage) during execution.

To add support for a new data store to the query engine, the database administrator must implement a new wrapper. Whether the new data store will be subqueried through CloudMdsQL with SQL or native expressions depends on the data store's native query mechanism. If the new data store is an RDBMS or a mapping between the data store's query interface and SQL statements exist, the data store can be queried with SQL expressions and its wrapper should be implemented to handle subquery plans by translating them to native queries (see Sects. 6.4.1 and 6.4.2). Otherwise, the data store must be queried with native expressions and the wrapper implementation should handle only native queries (see Sect. 6.4.3).

6.4.1 Querying SQL compatible NoSQL data stores

Since the data model of some NoSQL data stores (e.g. key-value or document databases) can be considered as a subset of the relational model, in most cases it is possible

to map simple SQL commands to native queries, without compromising the functionality. In fact, SQL-like languages are already commonly used with data stores based on the BigTable data model, e.g. CQL for Cassandra. For such data stores, the recommended approach for subquerying within CloudMdsQL is to use SQL table expressions against the data store, even though the data store does not natively support SQL.

Whenever an SQL table expression is used as a nested query against a data store, it is considered as a sub-select statement and hence is transformed into a sub-tree in the query execution plan. Thus, each SQL table expression can be subject to further transformations and may be possibly rewritten by the optimizer before submitted for execution to the data store. This allows the CloudMdsQL engine to perform optimizations of the global query execution plan (like pushing selections, projections, aggregations, and joins down the tree into sub-plans) or take advantage of bind joins, etc. However, besides pushing down operations, the query optimizer does not perform further optimization (such as operation reordering) on a sub-plan, because it will only be used for building the corresponding native query, which normally is supposed to be optimized by the data store's optimizer. Each sub-plan is then delivered to the corresponding wrapper, which interprets and transforms it to a native query in order to execute it against the data store using its native query mechanism.

6.4.2 SQL capabilities

In order to build executable subquery plans, the query planner must be aware of the capabilities of the corresponding data store to perform operations supported by the common data model. Therefore, the wrapper implementer must identify the subset of the common algebra that is supported by the data store. Thus the query planner can take the decision which parts of the global query plan can be handled locally by the data stores and which part should remain in the common query plan (see Sect. 6.1). For example, a MongoDB data store can perform selection operations—analogous to the document collection method find()—but is not able to perform joins. Being aware of that, the query planner can push selection operations down to the subquery plan, but will assign any join operation between MongoDB document collections to the common query plan.

The method to handle data source capabilities, proposed in [22], requires that the query engine serializes the subquery plan (or single operations from it) to a sentence of a specific language, that should be matched against a pattern, provided by the corresponding wrapper—if the validation succeeds, then the data store is capable of executing the subquery. Thus the query planner can determine the boundary between the common query plan the sub-plan that will be handled by the data store.

In CloudMdsQL, a similar approach is proposed which makes use of JSON schemas [15] as an instrument for the wrapper to express its data store's capabilities. To test the executability of a sub-plan (or a single operation) against a data store, the query planner serializes it to a JSON document that has to be validated against the JSON schema exposed by the wrapper. Below is an example of a capability JSON schema for a key-value data store that is capable only of performing selection operations involving comparisons on the 'key' attribute (only certain elements of the schema object are shown):

```
{
  "properties": {
    "op": { "type": "string", "pattern": "SELECT" },
    "tableref": { "type": "string" },
    "filter": { "$ref": "#/definitions/expression" }
  },
  "definitions": {
    "expression": { "oneOf": [
      { "$ref": "#/definitions/comparison" },
      { "$ref": "#/definitions/function" }
    ] },
    "comparison": { "properties": {
      "comp": { "type": "string", "pattern": "=|<|>|<=|>=|<>" },
      "lhs": { "properties": {
        "colref": { "type": "string", "pattern": "key" },
      },
      "rhs": { "type": "string" }
    } },
    "function": { "properties": {
      "func": { "type": "string", "pattern": "AND|OR" },
      "lhs": { "$ref": "#/definitions/expression" },
      "rhs": { "$ref": "#/definitions/expression" }
    } }
  }
}
```

Now let us consider the following subquery that is composed of two conjunctive selection conditions, each of which is tested against the capability specification. The result of the validation shows that condition #1 can be handled by a selection operation in the key-value data store and therefore it will be left in the subquery, while condition #2 doesn't pass the validation, and therefore will be pulled up in the common plan to be processed by to common query engine.

```
SELECT key, value FROM tbl WHERE key BETWEEN 10 AND 20 AND value > key
```

Condition #1: `key BETWEEN 10 AND 20` Validation: success	Condition #2: `value > key` Validation: failure

`{`
`  "op": "SELECT",`
`  "tableref": "tbl",`
`  "filter": {`
`  "func": "AND",`
`  "lhs": { "comp": ">=",`
`  "lhs": {"colref": "key"},`
`  "rhs": "10" },`
`  "rhs": { "comp": "<=",`
`  "lhs": {"colref": "key"},`
`  "rhs": "20" }`
`  }`
`}`
	`{`
	`  "op": "SELECT",`
	`  "tableref": "tbl",`
	`  "filter": {`
	`  "comp": ">",`
	`  "lhs": {"colref": "value"},`
	`  "rhs": {"colref": "key"}`
	`  }`
	`}`

6.4.3 Using native queries

In a CloudMdsQL query, to write native named table expression subqueries against SQL incompatible data stores, embedded blocks of native query invocations are used. In such occasions, the wrapper is thin—it just takes the subquery as is and executes it against the data store without having to analyze the subquery or synthesize it from a

query plan. Thus the wrapper provides transparency allowing CloudMdsQL queries to take the most of each data store's native query mechanism. When the data store does not have a text-based native query language but offers only an API, the wrapper is expected to expose such API through an embedded scripting language. This language must fulfill the following two requirements:

- Each query must produce a relation according to the common data model; the corresponding wrapper is then responsible to convert the data set to match the declared signature, if needed.
- In order to fulfill the requirement for nested tables support, the language should provide a mechanism to instantiate and use data retrieved by other named table expressions.

In this paper we use Python as an example of embedded language used by a wrapper. The requirements above are satisfied by the `yield` keyword and `CloudMdsQL` object, similarly to what happens in Python named table expressions.

7 Use case example

To illustrate the details of CloudMdsQL query processing, we consider three databases (briefly referred to as DB1, DB2 and DB3) as follows:

DB1 is a relational (e.g. Derby) database storing information about scientists in the following table:
`Scientists:`

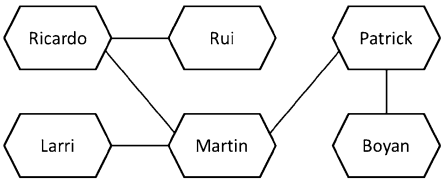
Name	Affiliation	Country
Ricardo	UPM	Spain
Martin	CWI	Netherlands
Patrick	INRIA	France
Boyan	INRIA	France
Larri	UPC	Spain
Rui	INESC	Portugal

DB2 is a document (e.g. MongoDB) database containing the following collections of publications and reviews:

```
Publications(
{id:1, title:'Snapshot Isolation', author:'Ricardo', date:'2012-11-10'},
{id:5, title:'Principles of DDBS', author:'Patrick', date:'2011-02-18'},
{id:8, title:'Fuzzy DBs',          author:'Boyan',   date:'2012-06-29'},
{id:9, title:'Graph DBs',          author:'Larri',   date:'2013-01-06'}
)

Reviews (
  {pub_id:1, reviewer: 'Martin',  date: '2012-11-18', review: '…text…'},
  {pub_id:5, reviewer: 'Rui',     date: '2013-02-28', review: '…text…'},
  {pub_id:5, reviewer: 'Ricardo', date: '2013-02-24', review: '…text…'},
  {pub_id:8, reviewer: 'Rui',     date: '2012-12-02', review: '…text…'},
  {pub_id:9, reviewer: 'Patrick', date: '2013-01-19', review: '…text…'}
)
```

DB3 is a graph database (e.g. Sparksee) representing a social network with nodes representing persons and 'friend-of' links between them:



We now reveal step by step how the following CloudMdsQL query is processed by the engine. The query involves all the three databases and aims to discover 'conflicts of interest in publications from Inria reviewed in 2013' (a conflict of interest about a publication is assumed to exist if the author and reviewer are friends or friends-of-friends in the social network). The subquery against DB3 uses the Sparksee Python API and user-defined functions and in particular, a function `FindShortestPathByName` defined over a graph object, which seeks the shortest path between two nodes by performing breadth-first search, referring the nodes by their 'name' attributes and limited to a maximal length of the sought path.

```
scientists( name string, affiliation string )@DB1 = (
  SELECT name, affiliation
  FROM scientists
)
pubs_revs( id int, title string, author string, reviewer string,
           review_date timestamp )@DB2 =
(
  SELECT p.id, p.title, p.author, r.reviewer, r.date
  FROM publications p, reviews r
  WHERE p.id = r.pub_id
)
friendships( person1 string, person2 string, level int
    JOINED ON person1, person2
    WITHPARAMS maxlevel int
    CARDINALITY = card(Outer)/2 )@DB3 =
{*
  for (p1, p2) in CloudMdsQL.Outer:
    sp = graph.FindShortestPathByName( p1, p2, $maxlevel )
    if sp.exists():
      yield (p1, p2, sp.get_cost())
*}
friendship_levels( level int, friendship string
    WITHPARAMS maxlevel int
    CARDINALITY = maxlevel ) =
{*
  for i in range(0, $maxlevel):
    yield (i + 1, 'friend' + '-of-friend' * i)
*}

SELECT pr.id, pr.title, pr.author, pr.reviewer, l.friendship
FROM scientists s, pubs_revs pr,
     friendships(2) f, friendship_levels(2) l
WHERE s.name = pr.author
  AND pr.author = f.person1 AND pr.reviewer = f.person2
  AND f.level = l.level
  AND pr.review_date BETWEEN '2013-01-01' AND '2013-12-31'
  AND s.affiliation = 'INRIA';
```

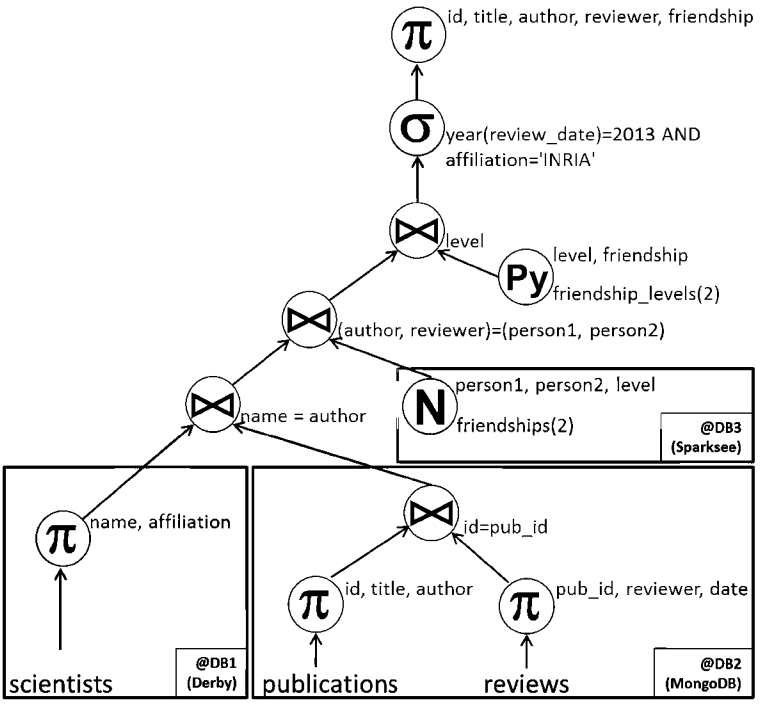


Fig. 3 Preliminary execution plan

This query contains two SQL subqueries—one against a relational database and the other against a document database. The parameterized native named table expression `friendships` against the graph database defines a relation that represents the level of friendship between two persons (expressed by the length of the shortest path between them). The parameter `maxlevel` indicates a maximal value for the length of the sought path; the expression is invoked with actual value of the parameter `maxlevel=2`, meaning that only relationships of type direct-friend or friend-of-friend will be found. The parameterized Python named table expression `friendship_levels` generates synthetic data containing textual representations of friendship levels between `1` and `maxlevel`. Both the native and Python expressions provide cardinality functions that will be used by the query planner to compare different query execution plans. The main select statement specifies the join operations to integrate data retrieved from the three data stores. Upon query decomposition the query planner prepares the preliminary execution plan shown on Fig. 3.

In this notation the rectangles denote the boundary between the common QEP and the sub-plans that are delivered to the wrappers for execution against data stores. Each operator is denoted by a circle with the operator symbol inside. The operator symbols **`N`** and **`Py`** correspond to the native expression and Python operator respectively. In subscript to each operation, additional information is specified, such as the name of the expression for native/Python operations, and the filter/join condition for selection/join

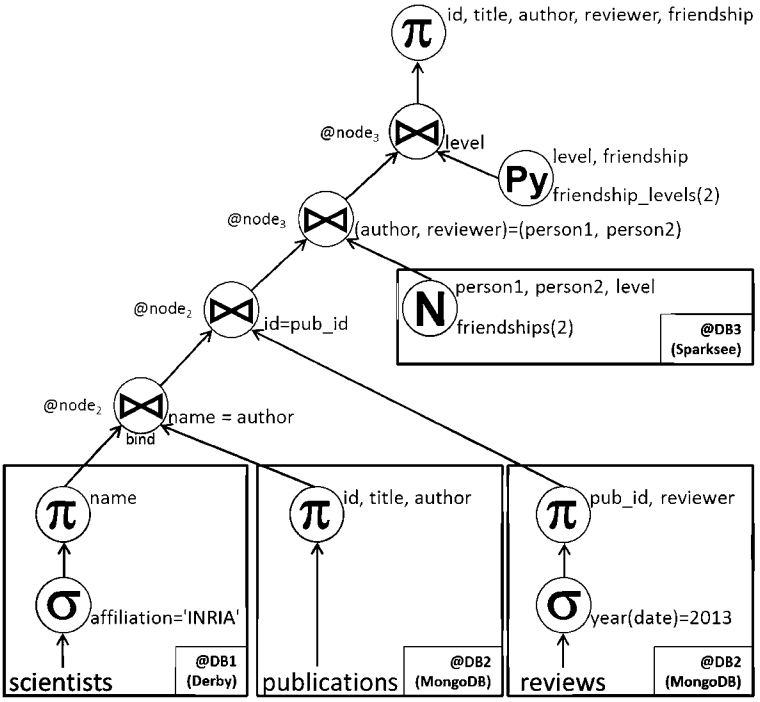


Fig. 4 Optimized execution plan

operations. In superscript, the columns of the corresponding intermediate relation are specified.

In the next step, the query planner verifies the executability of sub-plans against the capability specifications provided by each wrapper. First, it finds out that the MongoDB data store DB2 is not capable of performing the join between `publications` and `reviews`, therefore, it splits the sub-tree against DB2 into two sub-trees, aiming at independent retrieval of the two relations, and pulls the join operation in the common execution plan to be executed by the common query engine. Next, the optimizer seeks for opportunities for selection pushdowns, coordinating them as well with data store's capabilities. Thus, the selection `s.affiliation = 'INRIA'` is pushed into the sub-tree for DB1 and the selection `pr.review_date BETWEEN '2013-01-01' AND '2013-12-31'` is pushed into the sub-tree for DB2 that has to retrieve data from `reviews`. Doing this, the optimizer determines that the columns `s.affiliation` and `pr.review_date` are no longer referenced in the common execution plan, so they are simply removed from the corresponding projections on `scientists` and `reviews` from DB1 and DB2.

We assume that the Derby and MongoDB wrappers export the needed by the query optimizer metadata to the query engine's catalog. Taking also into account the cardinalities estimated by the user-defined cost functions of the native and Python expressions, the query planner searches for an optimal QEP, considering the usage of bind joins, join ordering, and the worker nodes in charge of each operation (which defines the way of shipping intermediate data). At the end of the optimization step, the prelimi-

nary plan is transformed into the plan on Fig. 4 that is passed to the query execution controller of $node_3$.

Each join operation in the QEP is supplemented with the identifier of the node that is in charge of executing it. The enumeration of the nodes is according to the indexes of the collocated data stores as we named them, i.e. $node_1$ is collocated with DB1, etc. The join between `scientists` and `publications` is marked with the label `bind`, which means that a bind join method will be performed.

The QEP is executed by performing the following steps, including the sequence of queries executed against the three data stores:

1. The Derby wrapper at $node_1$ sends the following SQL statement to retrieve data from the `scientists` table in the Derby database DB1, retrieves the corresponding intermediate relation, and transfers it to the operator engine of $node_2$:

```
SELECT name
FROM scientists
WHERE affiliation = 'INRIA'
```

Name
Patrick
Boyan

While retrieving the above tuples to the operator engine, the latter stores them in its temporary table storage and builds a set of distinct values of the column `name`, necessary for the next step.

2. The MongoDB wrapper at $node_2$ prepares a native query to send to the MongoDB database DB2 to retrieve those tuples from `publications` that match the bind join criteria. It takes into account the bind join condition derived from the already retrieved data from DB1 and generates a MongoDB query whose SQL equivalent would be the following:

```
SELECT id, title, author FROM publications
WHERE author IN ('Patrick', 'Boyan')
```

However, the wrapper for DB2 does not generate an SQL statement, instead it generates directly the corresponding MongoDB native query:

```
db.publications.find(
  { author: {$in:['Patrick', 'Boyan']} },
  { id: 1, title: 1, author: 1, _id: 0 }
)
```

Upon receiving the result dataset (a MongoDB document collection), the wrapper converts each document to a tuple, according to the signature of the named table expression `pubs_revs`, and then pipelines the tuples to the operator engine, which performs the actual join operation using the already retrieved result set from step 1. The result of the bind join is the contents of the following intermediate relation:

id	Title	Author
5	Principles of DDBS	Patrick
8	Fuzzy DBs	Boyan

Since this relation will be consumed by only one operator, the operator engine does not need to store it in the table storage; therefore the tuples are simply pipelined as input to the operation described in step 4.

3. Independently from steps 1 and 2, the wrapper prepares another MongoDB query for DB2 that, taking into account the pushed down selection, retrieves reviews made in 2013. The generated native query (preceded by its SQL equivalent) and the result intermediate relation are as follows:

```
SELECT pub_id, reviewer FROM reviews
 WHERE date BETWEEN '2013-01-01' AND '2013-12-31'

db.reviews.find(
  { date: {$gte:'2013-01-01', $lte:'2013-12-31'} },
  { pub_id: 1, reviewer: 1, _id: 0 }
)
```

Pub_id	Reviewer
5	Rui
5	Ricardo
9	Patrick

4. The intermediate relations from steps 2 and 3 are joined by the operator engine at $node_2$ to result in another intermediate relation, which is transferred to the operator engine of $node_3$ to be pipelined to the next join operator:

id	Title	Author	Reviewer
5	Principles of DDBS	Patrick	Rui
5	Principles of DDBS	Patrick	Ricardo

5. The query engine sends to the wrapper of DB3 the Python code to be executed against the graph database. It also provides an entry point to the intermediate data, represented by the special Python object `CloudMdsQL`. The wrapper of DB3 has preliminarily initialized the object `graph`, needed to reference the database's graph data. The Python code of the named table expression `friendships` uses the iterator `Outer` to iterate through a projection on the join attribute columns of the other side of the join, in which the named table participates, namely the tuples pipelined from the intermediate relation of step 4. For each tuple it tests if there exists a path with maximal length `maxlevel=2` between the author and reviewer in the graph database. The produced tuples are as follows:

Person1	Person2	Level
Patrick	Ricardo	2

As the above tuples are generated by the Python expression `friendships`, they are immediately joined with their corresponding tuples of the relation from step 4 to produce the next intermediate relation:

id	Title	Author	Reviewer	Level
5	Principles of DDBS	Patrick	Ricardo	2

6. Independently from all of the above steps, the operator engine at $node_3$ executes the Python code of the expression `friendship_levels`, instantiated with parameter value `maxlevel=2` to produce the relation:

Level	Friendship
1	friend
2	friend-of-friend

Essentially, the involvement of this Python operator is not needed for the purpose of the query, because the textual representation of a level of friendship can be generated directly within the code of the native expression `friendships`. However, we include it in the example in order to demonstrate a wider range of CloudMdsQL operators.

7. Finally, the root join operation is performed, taking as input the pipelined tuples of the intermediate relation from step 5 and matching them to the one from step 6, to produce the final result:

id	Title	Author	Reviewer	Friendship
5	Principles of DDBS	Patrick	Ricardo	friend-of-friend

This use case example demonstrates that the proposed query engine achieves its objectives by fulfilling the five requirements as follows:

1. It preserves the expressivity of the local query languages by embedding native queries, as it was demonstrated with the named table expression `friendships`.
2. It allows nested queries to be chained and nesting is allowed in both SQL and native expressions, as it was demonstrated in two scenarios. First, the subquery against the MongoDB database DB2 uses as input the result from the subquery to the relational database DB1. Second, the subquery against the Sparksee graph database DB3 iterates through data retrieved from the other two databases.
3. The proper functioning of the query engine does not depend on the data stores' schemas; it simply converts the data retrieved from data stores to match the ad-hoc schema defined by the named table expressions' signatures.
4. It allows data-metadata transformations as it was demonstrated with the named table expression `friendships`: metadata (the length of a path in the graph database) is converted to data (the level of friendship). It also allows data to be synthesized as with the Python table expression `friendship_levels`.
5. It allows for optimizing the query execution by rewriting subqueries according to the bind join condition and the pushed down selections and planning for optimal join execution order and intermediate data transfer.

8 Experimental validation

The goal of our experimental validation is to show the ability of the query engine to optimize CloudMdsQL queries, as optimizability is one of the objectives of the query

language. Notice that our experiments are not intended for benchmarking the query engine; their purpose is to illustrate the impact of each optimization technique on the overall efficiency of the query execution. To achieve this, we have implemented the first prototype of our query engine, aiming at implementing all the proposed optimization techniques, while giving less importance to the efficiency of the operator engine. In this section, we first describe the current implementation of the query engine prototype. Then, we introduce the datasets, based on the use case example in Sect. 7. Finally, we present our experimental results.

8.1 Prototype

For the current implementation of the query engine, we modified the open source Derby database to accept CloudMdsQL queries and transform the corresponding execution plan into Derby SQL operations. We developed the query planner and the query execution controller and linked them to the Derby core, which we use as the operator engine. The main reasons to choose Derby database to implement the operator engine are because Derby:

- Allows extending the set of SQL operations by means of CREATE FUNCTION statements. This type of statements creates an alias, which an optional set of parameters, to invoke a specific Java component as part of an execution plan.
- Has all the relational algebra operations fully implemented and tested.
- Has a complete implementation of the JDBC API.
- Allows extending the set of SQL types by means of CREATE TYPE statements. It allows working with dictionaries and arrays.

Having a way to extend the available Derby SQL operations allows designing the resolution of the named table expressions. In fact, the query engine requires three different components to resolve the result sets retrieved from the named table expressions:

- `WrapperFunction`: To send the partial execution plan to a specific data store using the wrappers interfaces and retrieve the results.
- `PythonFunction`: To process intermediate result sets using Python code.
- `NestedFunction`: To process nested CloudMdsQL queries.

Named table expressions admit parameters using the keyword `WITHPARAMS`. However, the current implementation of the `CREATE FUNCTION` statement is designed to bind each parameter declared in the statement with a specific Java method parameter. In fact, it is not designed to work with Java methods that can be called with a variable number of parameters, which is a feature introduced since Java 6. To solve this gap, we have modified the internal validation of the `CREATE FUNCTION` statement and how to invoke Java methods with a variable number of parameters during the evaluation of the execution plan. For example, imagine that the user declares a table named expression `T1` that returns 2 columns (`x` and `y`) and has a parameter called `a` as follows:

```
T1( x int, y string  WITHPARAMS a string )@db1 =
( SELECT x, y FROM tbl WHERE id = $a )
```

The query execution controller will produce dynamically the following `CREATE FUNCTION` statement:

```
CREATE FUNCTION T1 ( a VARCHAR( 50 ) )
RETURNS TABLE ( x INT, y VARCHAR( 50 ) )
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
READS SQL DATA
EXTERNAL NAME 'WrapperFunction.execute'
```

It is linked to the following Java component, which will use the wrapper interfaces to establish a communication with the data store `db1`:

```
public class WrapperFunction {
     public static ResultSet execute(
                         String namedExprName,
                         Long queryId,
                         Object... args /*dynamic args*/) throws Exception {
          //Code to invoke the wrappers
     }
}
```

Therefore, after accepting the execution plan, which is produced in JSON format, the query execution controller parses it, identifies the sub-plans within the plan that are associated to a named table expression and dynamically executes as many `CREATE FUNCTION` statements as named table expressions exist with a unique name. As a second step, the execution engine evaluates which named expressions are queried more than once and must be cached into the temporary table storage, which will be always queried and updated from the specified Java functions to reduce the query execution time. Finally, the last step consists of translating all operation nodes that appear in the execution plan into a Derby specific SQL execution plan. In fact, this is the same result that Derby originally produces when parses an SQL query. Once the SQL execution plan is valid, the Derby core (which acts as the operator engine) produces a dynamic byte code that resolves the query that can be executed as many times as the application needs.

Derby implements the JDBC interface and an application can send queries though the Statement class. So, when the user has processed the query result and closed the statement, the query execution controller drops the previously created functions and cleans the temporary table storage.

The rest of the query engine components are developed as follows:

- The query planner is implemented in C++ and uses the Boost.Spirit framework for parsing context-free grammars, following the recursive descent approach.
- The wrappers are Java classes implementing a common interface used by the operator engine to interact with them.

We use three data stores—Sparksee (a graph database with Python API), Derby (a relational database accessed through its Java Database Connectivity (JDBC) driver) and MongoDB (a document database with a Java API). To be able to embed subqueries against these data stores, we developed wrappers for each of them as follows:

- The wrapper for Sparksee accepts as raw text the Python code that needs to be executed against the graph database using its Python client API in the environment of a Python interpreter embedded within the wrapper.

- The wrapper for Derby executes SQL statements against the relational database using its JDBC driver. It exports an `explain()` function that the query planner invokes to get an estimation of the cost of a subquery. It can also be queried by the query planner about the existence of certain indexes on table columns and their types. The query planner may then cache this metadata information in the catalog.
- The wrapper for MongoDB is implemented as a wrapper to an SQL compatible data store, i.e. it performs native MongoDB query invocations according to their SQL equivalent. The wrapper maintains the catalog information by running probing queries such as `db.collection.count()` to keep actual database statistics, e.g. cardinalities of document collections. Similarly to the Derby wrapper, it also provides information about available indexes on document attributes.

8.2 Datasets

We performed our experimental evaluation in the context of the use case example, presented in Sect. 7. For this purpose, we generated data to populate the Derby table `scientists`, the MongoDB document collections `publications` and `reviews`, and the Sparksee graph database with scientists and friendship relationships between them. All data is uniformly distributed and consistent. The datasets have the following characteristics:

- Table `scientists` contains 10k rows, distributed over 1000 distinct affiliations, thus setting to 0.1% the selectivity of an arbitrary equality condition on the `affiliation` attribute.
- Collection `publications` contains 1M documents, with uniform distribution of values of the `author` attribute, making 100 publications per scientist. The total size of the collection is 1GB.
- Collection `reviews` contains 4M documents, making 4 reviews per publication. The `date` attribute contains values between 2012-01-01 and 2014-12-31. This sets to 33% the selectivity of the predicate `year(date) = 2013`. The `review` attribute contains long string values. The total size of the collection is 20GB.
- The graph database contains one node per scientist and 500k edges between them. This data is generated to assure that for each publication, 2 out of 4 reviewers are friends or friend-of-friends to the author.
- The catalog contains sufficient information, collected through the Derby and MongoDB wrappers, about the above specified cardinalities and selectivities. It also contains information about the presence of indexes on the attributes `scientists.affiliation`, `publications.id`, `publications.author`, `reviews.pub_id`, `reviews.reviewer`, and `reviews.date`.

8.3 Experiments

We loaded the generated datasets in 4 data stores, each running on a separate node in a cluster, as follows: Apache Derby at node_1 stores the `scientists` table, MongoDB

at $node_2$ and $node_3$ stores respectively the `publications` and `reviews` document collections, and the Sparksee graph database at $node_4$. The data store identifiers that we use within our queries are respectively `DB1`, `DB2`, `DB3`, and `DB4`. Each node in the cluster runs on a quad-core CPU at 2.4GHz, 32 GB main memory, 1.5Gbps HDD throughput, and the network bandwidth is 1Gbps.

To demonstrate in detail all the optimization techniques and their impact on the query execution, we prepared 5 different queries. For each of them, we chose 3 alternative QEPs to run and compare their execution times, with different join orders, intermediate data transfer, and subquery rewritings. The execution times for the different QEPs are illustrated in each query's corresponding graphical chart.

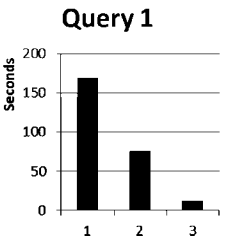
All the queries use the following common named table expressions, which we created as stored expressions:

```
CREATE NAMED EXPRESSION
scient( name string, affiliation string )@DB1 = (
  SELECT name, affiliation FROM scientists
);
CREATE NAMED EXPRESSION
pubs( id int, title string, author string )@DB2 = (
  SELECT id, title, author FROM publications
);
CREATE NAMED EXPRESSION
revs( pub_id int, reviewer string, date timestamp, review string )@DB3 =
(
  SELECT pub_id, reviewer, date, review FROM reviews
);
CREATE NAMED EXPRESSION
friends( name string, friend string JOINED ON name
    CARDINALITY = 100*card(Outer) )@DB4 =
{*
  for n in CloudMdsQL.Outer:
    for f in graph.GetNeighboursByName( n ):
      yield ( n, f.getName() )
*};
CREATE NAMED EXPRESSION
friendships( person1 string, person2 string, friendship string
    JOINED ON person1, person2   WITHPARAMS maxlevel int
    CARDINALITY = card(Outer) )@DB4 =
{*
  for (p1, p2) in CloudMdsQL.Outer:
    sp = graph.FindShortestPathByName( p1, p2, $maxlevel )
    if sp.exists():
      yield ( p1, p2, 'friend' + '-of-friend' * (sp.get_cost()-1) )
*};
```

Thus, each of the queries is expressed as a single SELECT statement that uses the above named table expressions. For each of the queries we describe the alternative QEPs with a text notation, using the special symbols $\ltimes$ for joins, $\bowtie$ for bind joins (where the join condition is bound to the right side of the join), $\sigma()$ for selections, and **@** in subscript to denote the node at which the operation is performed. If a selection is marked with **@QE** in subscript, then it is performed by the query engine, otherwise it is pushed down to be executed by the data store. The operation order is specified explicitly using parentheses. The relations within the QEP are referred with their first letter in capital, e.g. **R** stands for `reviews`.

Query 1 involves 2 tables and focuses on selection pushdowns and bind joins. The selectivity of the WHERE clause predicate is approximately 0.1%, which explains the benefit of the pushed down selection in QEP_{12} that reduces significantly the data retrieved from the `reviews` document collection in `DB3`. Using a bind join in QEP_{13} reduces to 0.4% the data retrieved from the `publications` collection.

```
SELECT p.id, p.title, p.author,
       r.reviewer, r.review
FROM pubs p JOIN revs r ON p.id = r.pub_id
WHERE r.date = '2013-05-01'
```



The alternative query plans are:

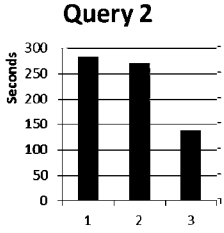
QEP_{11}: $\sigma_{@QE}(R) \bowtie_{@3} P$

QEP_{12}: $\sigma(R) \bowtie_{@3} P$

QEP_{13}: $\sigma(R) \ltimes_{@3} P$

Query 2 involves 3 tables and focuses on the importance of choosing the optimal data shipping direction. All the plans involve the retrieval and transfer of a selection (6GB) on the `reviews` collection and the entire `publications` collection (1GB). QEP_{21} retrieves both tables remotely. QEP_{22} retrieves `P` locally and `R` remotely. QEP_{23} retrieves `R` locally and $\sigma(S) \bowtie P$ (only 1MB) remotely. Although bind joins are applicable in all QEPs, we do not use them in order to focus on shipping of unfiltered data.

```
SELECT p.id, p.title, p.author, r.reviewer, r.review
FROM pubs p JOIN revs r ON p.id = r.pub_id
  JOIN scient s ON s.name = p.author
WHERE r.date BETWEEN '2013-01-01' AND '2013-12-31'
  AND s.affiliation = 'affiliation1'
```



The alternative query plans are:

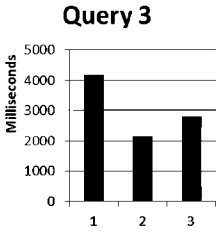
QEP_{21}: $(\sigma(S) \bowtie_{@1} P) \bowtie_{@1} \sigma(R)$

QEP_{22}: $(\sigma(S) \bowtie_{@2} P) \bowtie_{@2} \sigma(R)$

QEP_{23}: $(\sigma(S) \bowtie_{@2} P) \bowtie_{@3} \sigma(R)$

Query 3 involves 3 tables, of which the table `scientists` is used twice. To distinguish them, in the description of QEPs we use the symbols **`Sa`** and **`Sr`**. Because of the use of bind joins, this query handles much less data and executes much faster compared to the previous queries. The query focuses on different join orders, the effect of which comes mostly from the different selectivities of the bind join conditions.

```
SELECT p.id, p.title, p.author, r.reviewer, r.review, sr.affiliation
FROM pubs p JOIN revs r ON p.id = r.pub_id
  JOIN scient sa ON sa.name = p.author
  JOIN scient sr ON sr.name = r.reviewer
WHERE sa.affiliation = 'affiliation1' AND
  sr.affiliation IN ('affiliation2', 'affiliation3')
```



The alternative query plans are:

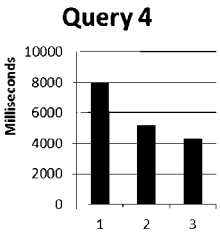
QEP_{31}: $((\sigma(Sr) \ltimes_{@3} R) \ltimes_{@3} P) \bowtie_{@3} \sigma(Sa)$

QEP_{32}: $((\sigma(Sa) \ltimes_{@2} P) \ltimes_{@3} R) \bowtie_{@3} \sigma(Sr)$

QEP_{33}: $(\sigma(Sa) \ltimes_{@2} P) \bowtie_{@3} (\sigma(Sr) \ltimes_{@3} R)$

Query 4 includes the `friendships` subquery against the graph database and focuses on the involvement of native named table expressions, using join iteration, and the usage of expensive native operations, such as breadth-first search. As the QEPs correspond to the ones for Query 3, the execution times depend on the join orders, but also on the number of distinct values of the relation to be joined with the `friendships` expression, which determines how many times breadth-first search is invoked.

```
SELECT p.id, p.title, p.author, r.reviewer,
       r.review, f.friendship
FROM pubs p JOIN revs r ON p.id = r.pub_id
  JOIN scient sa ON sa.name = p.author
  JOIN scient sr ON sr.name = r.reviewer
  JOIN friendships(2) f ON p.author = f.person1
                       AND r.reviewer = f.person2
WHERE sa.affiliation = 'affiliation1' AND
  sr.affiliation IN ('affiliation2', 'affiliation3')
```



The alternative query plans are:

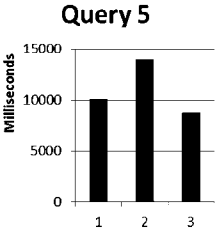
QEP_{41}: **(((σ(Sr) ⋈@3 R) ⋈@3 P) ⋈@3 F) ⋈@3 σ(Sa)**

QEP_{42}: **(((σ(Sa) ⋈@2 P) ⋈@3 R) ⋈@3 F) ⋈@3 σ(Sr)**

QEP_{43}: **((σ(Sa) ⋈@2 P) ⋈@3 (σ(Sr) ⋈@3 R)) ⋈@3 F**

Query 5 resembles Query 4, but uses the `friends` native subquery that invokes another native operation that yields many output tuples for a single input tuple. Like for Query 4, the join order determines when the native expression is invoked and the number of its input tuples.

```
SELECT p.id, p.title, p.author, r.reviewer,
       r.review, f.friend
FROM pubs p JOIN revs r ON p.id = r.pub_id
  JOIN scient sa ON sa.name = p.author
  JOIN scient sr ON sr.name = r.reviewer
  JOIN friends f ON r.reviewer = f.name
WHERE sa.affiliation = 'affiliation1' AND
  sr.affiliation IN ('affiliation2', 'affiliation3')
```



The alternative query plans are:

QEP_{51}: **(((σ(Sr) ⋈@3 R) ⋈@3 P) ⋈@3 F) ⋈@3 σ(Sa)**

QEP_{52}: **(((σ(Sa) ⋈@2 P) ⋈@3 R) ⋈@3 F) ⋈@3 σ(Sr)**

QEP_{53}: **((σ(Sa) ⋈@2 P) ⋈@3 (σ(Sr) ⋈@3 R)) ⋈@3 F**

9 Conclusion

In this paper, we proposed CloudMdsQL, a common language for querying and integrating data from heterogeneous cloud data stores and its query engine. By combining the expressivity of functional languages and the manipulability of declarative relational languages, it stands in "the golden mean" between the two major categories of query languages with respect to the problem of unifying a diverse set of data management systems. CloudMdsQL satisfies all the legacy requirements for a common query language, namely: support of nested queries across data stores, data-metadata transformations, schema independence, and optimizability. In addition, it allows embedded

invocations to each data store's native query interface, in order to exploit the full power of data stores' query mechanism.

The architecture of CloudMdsQL query engine is fully distributed, so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. Thus, the query engine does not follow the traditional mediator/wrapper architectural model where mediator and wrappers are centralized. This distributed architecture yields important optimization opportunities, e.g. minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node. The wrappers are designed to be transparent, making the heterogeneity explicit in the query in favor of preserving the expressivity of local data stores' query languages. CloudMdsQL sticks to the relational data model, because of its intuitive data representation, wide acceptance and ability to integrate datasets by applying joins, unions and other relational algebra operations.

To validate the common query language concepts, we presented the way CloudMdsQL query engine, implemented according to the specified design, achieves its objectives. In our validation setup, we integrated three database management systems—Sparksee (a graph database with Python API), Derby (a relational database accessed through its JDBC driver) and MongoDB (a document database with a Java API). By executing representative CloudMdsQL queries and revealing what the query engine does to process them, we showed that the common query language satisfies the five important requirements for a cloud multidatabase query language. In particular, it allows nested queries to be chained and nesting is allowed in both SQL and native expressions. Furthermore, it allows for optimizing the query execution by rewriting queries according to bind joins and pushed down selections, planning optimal join execution orders, and performing optimal shipping of intermediate data. Our experimental evaluation illustrates the impact of the used optimization techniques on the overall query execution performance.

CloudMdsQL has been extended to deal with distributed processing frameworks such as Hadoop MapReduce and Apache Spark [3]. It takes full advantage of the functionality of the underlying data processing frameworks by enabling the ad hoc usage of user defined map/filter/reduce operators in combination with traditional SQL statements, yet allowing for optimization by pushing down predicates and bind join conditions inside the map/filter/reduce chain to be applied as early as possible.

Acknowledgments Work partially funded by the European Commission through the CoherentPaaS FP7 Project funded under contract FP7-611068 [5]. We want to thank Norbert Martínez-Bazan for his contributions on the first version of the CloudMdsQL query engine. We also thank the editor and reviewers for their careful readings and useful suggestions that helped improving our design and the paper. The work of Prof. Ricardo Jimenez was also partially funded by the Regional Government of Madrid (CAM) under Project Cloud4BigData (S2013/ICE-2894) cofunded by ESF & ERDF, and the Spanish Research Council (MICCIN) under Project BigDataPaaS (TIN2013-46883).

References

1. Armbrust, M., Xin, R., Lian, C., Huai, Y., Liu, D., Bradley, J., Meng, X., Kaftan, T., Franklin, M., Ghodsi, A., Zaharia, M.: Spark SQL: Relational Data Processing in Spark. ACM SIGMOD Int. Conf. on Management of Data, pp. 1383-1394 (2015)

2. Binnig, C., Rehrmann, R., Faerber, F., Riewe, R.: FunSQL: It is time to make SQL functional. Int. Conf. on Extending Database Technology / Database Theory (EDBT/ICDT), pp. 41-46 (2012)
3. Bondiombouy, C., Kolev, B., Levchenko, O., Valduriez, P. : Integrating Big Data and Relational Data with a Functional SQL-like Query Language. Int. Conf. on Databases and Expert Systems Applications (DEXA), pp. 170-185 (2015)
4. Bugiotti, F., Bursztyn, D., Deutsch, A., Ileana, I., Manolescu, I.: Invisible Glue: Scalable Self-Tuning Multi-Stores. Conf. on Innovative Data Systems Research (CIDR), 7pp (2015)
5. CoherentPaaS Project, http://coherentpaas.eu. [Last accessed on August 18, 2015]
6. Danforth, S., Valduriez, P.: A FAD for Data-Intensive Applications. IEEE Trans. on Knowledge and Data Engineering **4**(1), 34–51 (1992)
7. Doan, A., Halevy, A., Ives, Z.: Principles of Data Integration. Morgan Kaufmann, (2012)
8. Godfrey, P., Gryz, J., Hoppe, A., Ma, W., Zuzarte, C.: Query rewrites with views for XML in DB2. IEEE Int. Conf. on Data Engineering, pp. 1339–1350 (2009)
9. Gulisano, V., Jiménez-Peris, R., Patiño-Martinez, M., Valduriez, P.: StreamCloud: A Large Scale Data Streaming System. IEEE Int. Conf. on Distributed Computing Systems (ICDCS), pp. 126-137 (2010)
10. Gulisano, V., Jiménez-Peris, R., Patiño-Martinez, M., Soriente, C., Valduriez, P.: StreamCloud: An Elastic and Scalable Data Streaming System. IEEE Trans. On Parallel and Distributed Systems **23**(12), 2351–2365 (2012)
11. Haas, L. M., Kossmann, D., Wimmers, E. L., Yang, J.: Optimizing Queries across Diverse Data Sources. Int. Conf. on Very Large Databases (VLDB), pp. 276-285 (1997)
12. Haase, P., Mathäß, T., Ziller, M.: An Evaluation of Approaches to Federated Query Processing over Linked Data. Int. Conf. on Semantic Systems (I-SEMANTICS) (2010)
13. Hacıgümüs, H., Sankaranarayanan, J., Tatemura, J., LeFevre, J., Polyzotis, N.: Odyssey: A Multi-Store System for Evolutionary Analytics. Proceedings of the VLDB Endowment (PVLDB) **6**(11), 1180–1181 (2013)
14. Hart, B., Valduriez, P., Danforth, S.: Parallelizing FAD using Compile Time Analysis Techniques. IEEE Data Engineering Bulletin (12) **1**, 9–15 (1989)
15. JSON Schema and Hyper-Schema, http://json-schema.org. [Last accessed on August 18, 2015]
16. LeFevre, J., Sankaranarayanan, J., Hacıgümüs, H., Tatemura, J., Polyzotis, N., Carey, M.: MISO: Souping Up Big Data Query Processing with a Multistore System. ACM SIGMOD Int. Conf. on Management of Data, pp. 1591-1602 (2014)
17. Liu, Z.H., Chang, H.J., Sthanikam, B.: Efficient support of XQuery Update Facility in XML enabled RDBMS. IEEE Int. Conf. on Data Engineering, pp. 1394–1404 (2012)
18. Martínez-Bazan, N., Muntés-Mulero, V., Gómez-Villamor, S., Águila-Llorente, M.A., Domínguez-Sal, D., Larriba-Pey, J-L.: Efficient Graph Management Based on Bitmap Indices. Int. Database Engineering & Applications Symposium (IDEAS), pp. 110-119 (2012)
19. Meijer, E., Beckman, B., Bierman, G. M.: LINQ: Reconciling Object, Relations and XML in the .NET Framework. ACM SIGMOD Int. Conf. on Data Management, pp. 706-706 (2006)
20. NoSQL Databases, http://nosql-database.org. [Last accessed on August 18, 2015]
21. Özsu, T., Valduriez, P.: Principles of Distributed Database Systems – Third Edition. Springer, 850 pages (2011)
22. Tomasic, A., Raschid, L., Valduriez, P.: Scaling Access to Heterogeneous Data Sources with DISCO. IEEE Transactions on Knowledge and Data Engineering **10**(5), 808–823 (1998)
23. Valduriez, P., Danforth, S.: Functional SQL, an SQL Upward Compatible Database Programming Language. Information Sciences **62**(3), 183–203 (1992)
24. Wyss, C.M., Robertson, E.L.: Relational Languages for Metadata Integration. ACM Trans. On Database Systems **30**(2), 624–660 (2005)
25. Yuanyuan, T., Zou, T., Özcan, F., Goncalves, R., Pirahesh, H.: Joins for Hybrid Warehouses: Exploiting Massive Parallelism and Enterprise Data Warehouses. Int. Conf. on Extending Database Technology / Database Theory (EDBT/ICDT), pp. 373-384 (2015)
26. Zhu, M., Risch, T.: Querying Combined Cloud-Based and Relational Databases, Int. Conf. on Cloud and Service Computing, pp. 330–335 (2011)
27. Zhu, Q., Larson, P.-A.: Global Query Processing and Optimization in the CORDS Multidatabase System, Int. Conf. on Parallel and Distributed Computing Systems, pp. 640–647 (1996)